Efficient and Effective Buffer Overflow Protection on ARM Processors

Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens

Efficient and Effective Buffer Overflow Protection on ARM Processors

Raoul Strackx, Yves Younan, Pieter Philippaerts, and Frank Piessens

Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium
raoul.strackx,yves.younan,pieter.philippaerts,
frank.piessens@cs.kuleuven.be

Abstract. Although many countermeasures have been developed for desktop and server environments, buffer overflows still pose a big threat. The same approach can be used to target mobile devices. Unfortunately, they place more severe limitations on countermeasures. Not only are the performance requirements at least as important, memory and power consumption need to be considered as well. Moreover, processors used in mobile devices generally are equipped with a different instruction set. Therefore countermeasures may not be ported easily.
Multistack is an effective countermeasure against stack-based buffer overflows. It protects applications by using multiple stacks to separate possible attack targets from possible sources. However, its performance overhead will no longer be negligible on the ARMv7 platform (widely used on mobile devices) and it wastes too much memory, making it too costly for mobile applications. We propose 3 methods to reduce memory overhead up to 28% with only a 3.91% performance overhead.

Key words: Control flow attacks, Stack-based buffer overflow, Software security, mobile platform

1 Introduction

Buffer overflow vulnerabilities pose a significant threat to applications written in unsafe languages and to the devices they run on. Most of the existing buffer overflow attacks write past the boundary of a buffer located on the stack and modify an interesting memory location. Function return addresses are a frequently chosen target of attack. By overwriting them with the location of code inserted as data into memory, the program can be forced to execute instructions with the privilege level of the attacked program [1, 2].

Many countermeasures that protect desktop and server environments have been developed. Some try to solve the problem entirely by inserting bound checks or modifying the language itself [3–5]. Others rely on randomness and secrets to detect or prevent modifications of data in memory. The latter have less performance overhead [6–9]. Even though many more approaches exist, attackers still find ways to attack systems successfully, for example by breaking some of the assumptions made in the countermeasures in place, such as the fact that a canary remains secret [10].

According to the NIST's National Vulnerability Database [11], 563 buffer overflow vulnerabilities were reported in 2008, making up 10% of the total 5,634 vulnerabilities reported in that period, only preceded by Cross Site Scripting (14.0%) and SQL injection vulnerabilities (19.5%). Of those buffer overflow vulnerabilities, 436 had a high severity rating. As a result, buffer overflows make up 15% of the 2,853 vulnerabilities with a high severity rating reported in 2008, second only to SQL injection vulnerabilities (28.5%).

During the last decade, mobile devices built upon the ARM architecture, such as smartphones and PDA's, became ubiquitous objects. They are used for a wide variety of tasks, ranging from surfing the web to managing important data. As their number increases, so does the interest of attackers in the platform. Since they are built using the same principle as desktop systems, they as well are vulnerable to buffer overflow attacks [12, 13].

However, defending embedded devices against these attacks is more difficult than desktop systems. To reduce costs and increase mobility, they are generaly equipped with a less powerfull processor and limited memory. Moreover, processors used in embedded devices (i.e. ARM CPU's) generally are equipped with a different instruction set than desktop systems. As a results, not all countermeasures can be ported easily.

Multistack [14] is an effective stack-based buffer overflow countermeasure, originally designed for the x86 architecture. It does not rely on secret values (such as canaries), but separates buffers from possible attack targets using guard pages. This prevents state-of-the-art attacks [10] with negligible performance overhead. However, its memory consumption is significant, making it too costly to protect applications on mobile devices. Moreover, it relies on the ability to add a 32-bit constant value to a register in a single load/write instruction, an operation not supported by ARM processors. As a result, performance degradates on this platform.

In this paper, three techniques are proposed to port the Multistack countermeasure to the ARMv7 platform. They all minimize memory consumption while at the same time performance overhead is reduced compared to the original Multistack method on the ARMv7 platform, making the countermeasure efficient enough to be deployed on existing mobile devices.

This paper is structured as follows: first buffer overflows are reexamined (section 2), followed by the original Multistack countermeasure. Section 4 presents the different approaches, which are evaluated in section 5. Possible enhancements are described in section 6. The proposed techniques are compared to existing countermeasures in section 7, while section 8 presents our conclusions.

2 Buffer overflows

Buffer overflows are the result of an out of bounds write operation on an array. In this section we briefly recap how an attacker could exploit such a buffer overflow. Many derivative attacks exists; more complete overviews can be found in [1, 2, 15].

Buffers can be allocated on the stack, the heap or in the data/bss section in C. For arrays[1] that are declared in a function body, space is reserved on the stack. Buffers that are allocated dynamically (using the *malloc* function, or some other variant), are put on the heap, while arrays that are global or static are allocated in the data/bss section. The array is manipulated by means of a pointer to the first byte. Bytes within the buffer can be addressed by adding the desired index to this base pointer.

```c
void copy(char* src, char* dst) {
  int i = 0;
  char curr = src[0];
  while(curr) {
    dst[i] = curr;
    i++;
    curr = src[i];
  }
}
```

Listing 1.1. A C function that is vulnerable to a buffer overflow.

Most C-compilers generate code that does not check the bounds of an array and allow programs to copy data beyond their end. This behavior can be used to overwrite data in adjacent memory space. The unprotected application can be successfully attacked if these memory locations contain data that influence control flow and is used after the buffer overflow.

On the stack this is usually the case: it stores the addresses to resume execution at, after a function call has completed its execution. This address is called the *return address*. Manipulating it gives the attacker the possibility to execute arbitrary code.

Listing 1.1 shows a straightforward string copy function. Improper use of this function can lead to a buffer overflow, because there is no validation that the destination buffer can actually hold the input string. An attacker can thus exploit the buffer overflow vulnerability to overwrite memory that is stored adjacent to the destination buffer.

3 Multistack

3.1 Approach

Multistack [14] is a separation-based countermeasure designed to protect applications against stack-based buffer overflows. For each data type that may be stored on the stack, an analysis is made to determine the feasibility that 1) it

[1] We will use *array* as a synonym for *buffer* throughout the paper.

can be used as a source of attack and 2) it will ever be a target of an attack. Using this information, types with a comparable source/target trade-off are placed in the same category.

For example, an array of characters is a common source of attack, but is rarely a target itself. Pointers on the other hand, are much more likely to be a target. In case an attacker is able to overwrite a pointer and specify the value written to memory by dereferencing that modified pointer, he/she can write an arbitrary value to a chosen memory location [16]. It is clear that these types should be separated from one another and they are placed in different categories.

This however, may not always be possible. Consider a structure containing an array of characters as well as pointers. This type may be used as an attack source, but may also be a target of attack. This type does not fit in either one of the two previous categories and a new one is created.

Multistack operates by placing the categories of variables on different stacks (see Figure 1). *Guard pages*[2] prevent buffer overflows on one stack from reaching another.

Note that the guard pages will not prevent arrays in structures to overwrite the accompanied pointers. However, in most applications the usage of such structures is modest and the chance of finding a buffer overflow vulnerability in code operating on them, is limited. Applications that do not fulfill this assumption may apply additional countermeasures, possibly only in functions that use such types as local variables.

By placing the separate stacks sequentially in memory, their exact location can be calculated at compile-time. Consider a variable `x` that should be placed on stack `s` with offset `d` from the stack pointer. When all stacks are assigned a maximum length of 8 MiB[3], the updated offset will be

$$fp - d - s \cdot 8MiB$$

where `fp`, the frame pointer, points to the start of the stack frame.

This has another advantage; it allows address space layout randomization (ASLR) (see section 7) to relocate the base of the stack each time the application is run. Obviously, the relative distance between the stacks must be specified at compile-time.

To address a variable on the stack, unprotected applications also have to perform this addition, though with a different operand. Since the x86 instruction set provides load and store instructions that are able to store an immediate constant of 32-bits, no extra instructions need to be issued to access the variable.

Multistack can be configured to use any number of stacks. However, when their number increases, the drawbacks of the countermeasure do as well (see section 3.2). Some methods that will be described in section 4 limit the maximum number of categories to four.

[2] A memory page without any permission. Any attempt to read, write or execute from this page will result in a segmentation fault.

[3] 1 MiB = 1 mebibytes = 2^{20} bytes (standardized by IEC 60027-2)

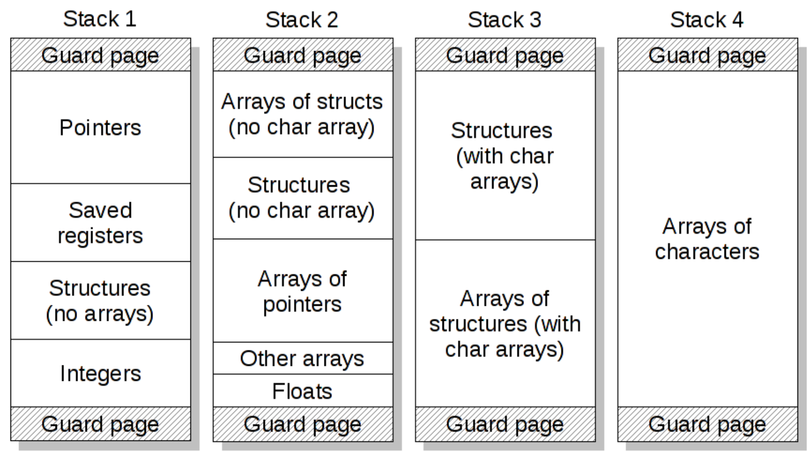

Fig. 1. Multistack stack layout for 4 stacks

3.2 Discussion

Performance penalty With the exception of code to allocate the different stacks and the creation of guard pages, no extra instructions are required. Performance evaluation [14] shows an overhead between 0% and 3%, on the x86 platform.

This efficiency is reached because this platform is able to use a 32-bit offset in a load and store instruction. On processors equipped with the ARM instruction set, multiple instructions would be required, leading to a degradation of performance (see section 4.1).

Memory consumption By only modifying the offset used to address a stack-based variable, performance loss remains negligible, but it also leads to wasted memory. The corresponding locations of a variable `x` on the other stacks, remain unused until the function returns. At any time in the execution of an application with `s` stacks and `m` the combined size of all variables on the stack, `(s - 1)m` MiB of virtual memory is wasted.

The situation even deteriorates in case multi-threaded applications are considered. In that case, each thread would waste that amount of memory. In section 5 a more detailed analysis is given.

3.3 Conclusion

The multistack countermeasure has a very limited performance penalty while providing effective protection against stack-based buffer overflows on the x86 platform. Its memory consumption and increased performance overhead on ARM processors, however make it too costly for mobile devices.

4 Approaches on mobile devices

4.1 Original approach

The most obvious implementation of Multistack on mobile devices is using a similar implementation as on the x86 platform. As described in section 3, a single frame pointer is used to address stack-based variables. This poses an issue since the ARMv7 instruction set does not support adding an arbitrary 32-bit integer using a single instruction. A simple solution is to use multiple additions (i.e. `add` instructions), however a performance penalty is expected.

The obvious drawback of this technique is its memory consumption. However, the evaluation in section 5 will reveal that this memory overhead in some cases is negligible.

4.2 Dedicated registers

Most applications never allocate space on the stack dynamically. Therefore the size of the stack frame is fixed for each function and the stack pointer can be used to address variables located on the stack. Applications that do not follow this assumption, can be modified by the compiler to not allocate space on the stack at runtime but on the heap instead.

ARM processors are equipped with a large number of registers to store intermediate results. To reduce memory consumption, three stack pointers can be stored in specially reserved registers and used to track the top of each stack.

This can be implemented in a compiler in a straightforward way. The `main` function is modified to allocate the different stacks and initialize the stack pointers. To each function's prologue and epilogue, instructions are added to allocate and free the required space on the related stacks by updating the corresponding registers. The addressing of stack-based variables is modified to use the appropriate stack pointer.

By reserving save-by-callee registers to hold the stack pointers, protected code is able to call unprotected libraries. However, the registers may not always contain the correct value when callbacks are used to return to protected code. Only in that case, recompilation of the library is required.

4.3 Indirection using a fixed address

The previous technique reserves three registers. This may lead to an increased register pressure and consequently a degradation of performance. Alternatively, the location of the stack pointers can be stored at a fixed location. Since all instructions in the ARMv7 instruction set have a fixed length of 32 bits, it is impossible to load an arbitrary 32-bit address in a register by specifying it as the payload of a single instruction. However, the `mov` instruction is able to do just that when certain requirements on the value are met. To be able to use this instruction, the stack pointers are stored on a reserved page at `0xbe000000 - 0xbe000008`.

Before executing any other code, the stacks are created and the page at `0xbe000000` is allocated. To allocate/deallocate memory on stack 2 to 4 or address a stack-based variable, the location of stack pointer 2 is loaded in a register first. The subsequent load or store instruction adds the required offset to access the relevant stack pointer. Note that by storing the stack pointers at a well chosen location, no register has to be reserved and only one extra instruction needs to be executed before each memory access related to stack 2 to 4.

4.4 Packed stack pointers

The previous technique assumes the increased register pressure to be the main contributor to the performance overhead of the countermeasure. In order to reduce it, stack pointers are loaded from and stored in memory when appropriate. This assumption however, may not hold for all applications. The presented "improvements" may in some cases even lead to a further deterioration of the countermeasure's performance.

Technological developments during the last decades resulted in a huge increase of processor and memory speed. However, making inexpensive, large memory that can be accessed rapidly, proved to be challenging. Multiple levels of caches were introduced to increase the overall performance of the system but accessing registers remains faster than loading a value from cache.

To load instructions from memory quickly, processors are equipped with pipelines to decode instructions. This technique works well when branches are reduced to a minimum. Considering both optimizations, it is obvious that it may be more efficient to execute more instructions when this avoids memory accesses.

The approach presented in this section uses another observation. The dedicated register approach uses one register for each stack pointer. While theoretically the stack is able to grow arbitrarily large, in practice only in extremely rare cases more than 8 MiB is required.

By aligning the added stacks at 8 MiB, the 11 most significant bits of their stack pointers will remain fixed for the entire execution of the program. As a result, by allocating the stacks at a location specified at compile time, only the 23 least significant bits need to be stored. This number can be reduced to 21 when it is assumed that each variable on the stack is 4-byte aligned. Using this approach [17], three stack pointers can be stored in only two registers. The stack pointer of the least frequently accessed stack will be split over the two registers.

Note that even more stack pointers can be stored using less registers by limiting the size of the stacks. Another option is to increase the alignment of both the stacks and the stored variables to facilitate larger stacks. However, in case a stack frame's length is not a multiple of the alignment, memory will be wasted.

As before, the `main` function is modified to allocate memory for the different stacks, to set the guard pages and initialize the stack pointers. Allocating space on a stack can be done without completely restoring the stack pointers. However, care must be taken when the stack pointer that is split over two registers needs to be updated. A carry bit may need to be added or subtracted from the most significant part of the pointer when the least significant part wraps around zero. By carefully choosing bit shift operations, this can be done without using any extra registers or branches.

Since this approach requires that two registers are reserved, the same incompatibility with callback functions as described in section 4.2 exists and a full recompilation of the application may be required.

5 Evaluation

During development, we focused mainly on smart phones and PDA's. The telecom and consumer suite of the MiBench benchmark [18] provides representative applications for these devices. They are used to evaluate memory and performance overhead. The `typeset` application is omitted because its in-line assembly uses the reserved registers. Applications that would only use stack 1 (CRC32 and adpcm) do not benefit from the countermeasure and are not considered.

5.1 Security evaluation

Two of the presented approaches store the stack pointers in registers. This has the advantage that an attacker is not able to modify this control data, before a successful code execution attack. In case the stack pointers are stored in memory (section 4.3), they are an interesting target of attack. Additional countermeasures need to be installed. However, a simple solution is to modify the location of stack 1 so it contains the memory locations where the pointers are stored.

With the exception of the packed stack pointers approach (section 4.4), the presented techniques have an additional advantage: they allow the locations of the stacks to be randomized, also relative to each other.

In all other aspects, our proposed approaches are as secure as the original Multistack countermeasure [14].

5.2 Memory consumption

The question how much physical memory can be gained by applying one of the proposed techniques for a specific application, is difficult to answer because many factors must be considered. The core of the problem is that when a virtual page remains unused, there is no need to store the corresponding physical page.

To determine the likelihood of this situation, we ran the telecom and consumer suite of MiBench, protected with the original Multistack implementation. Memory usage of each application was tracked, as well as the number of physical memory pages that were referenced.

These measurements relied on two assumptions. First, we assumed that in any function, the stack variables on stack 1 were placed closest to the base of the stack. Variables on stack 2 came next and so on. Note that the C standard does not specify the exact order of these variables in memory. Second, each stack, including stack 1, started at the beginning of an empty page. This is a worst case scenario. Applications with maximum stack sizes smaller than a single page, will not waste physical memory. In practice, memory may be wasted much sooner.

Table 1 displays the result of the test. Using these figures, the number of pages that will be gained by the proposed approaches, can be calculated easily. The measurements show improvements between 0% and 28% for the test applications, assuming a page size of 4,096 bytes. This large difference in potential gain depends on several factors, being discussed next.

Number, size and type of stack variables As explained in section 3.1, the original Multistack implementation, will result in holes of unused virtual memory. Depending on the number, size and type of stack variables, this in turn leads to internally fragmented, unused and/or completely used pages. Figure 2 displays these possibilities for the *lame* and *qsort* applications.

In case pages remain unused, no physical memory is wasted. This situation occurs when a stack is rarely used or when large arrays are allocated. When space is allocated to hold an array larger than two pages, at least one page remains unreferenced on the other stacks. The *qsort* application allocates $1.2 \cdot 10^6$ MiB to hold the values to be sorted and thus is a good example of this situation.

Compacting the stacks will have a positive influence in case many pages are internally fragmented. Consider stack 1 of the *lame* application. It uses 7 pages (28,672 bytes) to store only 1,444 bytes, or 35.3% of the memory it allocated. Figure 2 also displays the percentage of actually used physical memory to the amount that was allocated.

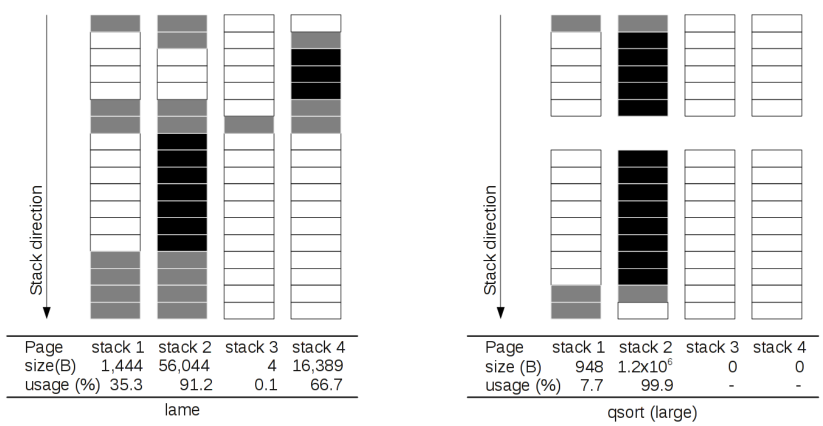

Fig. 2. Depending on the number, size and other factors, physical pages may remain unused (white boxes), internally fragmented (gray boxes) or completely used (black boxes). Compacting the stack minimizes the number of internally fragmented pages.

Number of function calls The number of function calls also plays an important role. Obviously, in case the number of calls is small, the combined stack size

Table 1. From the MiBench benchmark the telecom and consumer suites were selected. The different applications from these suites were run protected with the original Multistack implementation. This table displays the maximum number of memory locations used on the different stacks. The size in bytes is displayed together with the used physical pages. The last column displays how much memory is saved by compacting the stacks.

		stack 1		stack 2		stack 3		stack 4		Total		Potential gain		
application	set	bytes	pages	bytes	pages	bytes	pages	bytes	pages	bytes	pages	bytes	pages	%
qsort	small	796	2	0	0	7,680,000	0	0	0	7,680,796	1,878	8,192	2	0.11%
qsort	large	948	3	1,200,000	293	0	0	0	0	1,200,948	296	8,192	2	0.68%
FFT	both	360	1	48	1	0	0	0	0	408	2	0	0	0%
IFFT	both	360	1	48	1	0	0	0	0	408	2	0	0	0%
gsm (enc.)	both	544	1	957	1	0	0	0	0	1,501	2	0	0	0%
gsm (dec.)	both	540	1	932	1	0	0	0	0	1,472	2	0	0	0%
jpeg (enc.)	both	436	1	2,960	1	132	1	257	1	3,785	4	0	0	0%
jpeg (dec.)	both	480	1	1,484	1	132	1	257	1	2,353	4	0	0	0%
lame	both	1,444	7	56,044	15	4	1	16,389	6	73,881	29	32,768	8	27.59%
mad	small	1,216	3	7,192	1	632	1	27,674	8	36,714	15	16,384	4	26.67%
mad	large	1,216	2	368	1	404	1	27,674	8	29,662	12	8,192	2	16.67%
tiff2bw	both	472	1	100	1	0	0	1,104	1	1,676	3	0	0	0%
tiff2rgba	both	464	1	102	1	0	0	1,024	1	1,590	3	0	0	0%
tiffdither	both	1,044	1	92	1	0	0	1,104	1	2,240	3	0	0	0%
tiffmedian	both	532	1	228	1	0	0	80	1	840	3	0	0	0%

may not be larger than a single page. As a result, even though the original implementation does leave unused memory locations on the stacks, no pages can be saved by compacting the stacks. A number of examples of this situation can be found in the MiBench benchmark, including jpeg, tiff2bw, gsm, FFT, ...

The number of function calls made throughout the execution of the application and the way these calls are made, can play a role in memory usage. When functions call each other to a great depth, the size of the stack could grow large. As a result, multiple pages could be used to store only a few bytes.

Also, if a lot of consecutive calls are made, memory usage could degrade. The diversity of the different calls will decrease the chances that pages on other stacks remain unused.

Page size Consider the example that a variable is stored on a certain stack. When this variable is larger than two pages, the corresponding location at the other stacks may remain unused and no physical pages need to be stored. The chances that this situation occurs increases inversibly with the size of the pages.

Small pages have a second advantage; on average only half of the pages before and after such a hole are used. In case the pages are smaller, less physical memory is wasted.

Our memory usage measurement on the MiBench benchmark assumes that the stack pointer points to the beginning of a blank page when the Multistack countermeasure is installed. In practice this may not be the case. In the worst case, it points to the end. As a result, only a few bytes can be stored on the first page of stack 2 to 4. However, the original implementation of Multistack could be adjusted easily. By allocating more space on stack 1, memory consumption on the other stacks may decrease with one page.

Programming style It is clear that the way a program is written will have a huge impact on the structure of the stack. A programmer could opt solving a problem iteratively instead of using recursive calls. This would prevent the size of the stacks from growing rapidly and wasting memory.

Also by allocating large variables on the heap, more pages could be reused on the stack with less fragmentation. This could be implemented in a compiler as a heuristic.

5.3 Performance evaluation

To evaluate the performance of the different Multistack implementations, we ran the telecom and consumer suite of the MiBench benchmark [18] on a Sharp PC-z1 running Ubuntu Linux 9.04 (kernel 2.6.28). This netbook is equipped with an ARM Cortex-A8 CPU running at 800 MHz and 512 MiB RAM. The countermeasures were implemented in the gcc-4.3.3 revision 143643 compiler. The benchmark was compiled with `-O0 -fomit-frame-pointer` flags. To receive accurate results, each application was provided the large input set and run 500 times. Table 2 displays the results.

Explaining and predicting performance overhead is complex since there are many factors to consider. Two of the most relevant factors are discussed next.

Increased number of instructions There are two obvious places where instructions may be added. First, all techniques, with the exception of the original Multistack, require the prologues and epilogues to be modified to allocate memory on the stack. Obviously, this will have a negative impact on performance in case an application makes many function calls, for example, if recursion is used.

Second, additional instructions may be required to access a variable on a stack. Applications such as *fft* and *ifft* read and write to stack 2 very often. Hence, the time to access this stack will have a huge impact on their performance.

In case of the original Multistack implementation, the immediate offset in the load/store instruction can not be used to load/store a value on stack 2, 3 or 4 since this offset is too large. Therefore an extra addition is required, leading to a performance loss of 15.1% for *fft* and even 19.08% for *ifft*. In case dedicated registers are used to store the different stack pointers, the stack-based variables can be accessed in only one instruction. This leads to a much lower performance loss; 6.99% for *ftt* and 9.03% for *ifft*.

In case the stack pointers are packed, the accessed variable's location will have an influence as well. Acesses to stack 2 or 4 only require a few instructions. Variables on stack 3 are much harder to access since their stack pointer is scattered over 2 registers. As a result, accessing stack 3 is slower than stack 2 or 4. With the exception of the dedicated registers approach, the same applies for the other implementations; variables on stack 1 can be addressed by simply specifying an offset to register `sp`. For other stacks, more instructions are needed.

When main memory needs to be accessed, the order wherin the instructions are scheduled and the memory's access time will also play an import role.

Register pressure The dedicated register approach does not require additional instructions to access a stack-based variable. Therefore, it was able to execute the *fft* and *ifft* applications much faster. However, by reserving three registers, register pressure may increase, leading to a performance deterioration. To discover the impact on the selected applications, each was run unprotected with 3 reserved registers but no significant impact on performance was found.

Summary Many factors influence the performance of the different presented techniques. On average (see table 2), all the presented techniques perform better than the original Multistack implementation. By storing the stack pointers at a fixed location, performance overhead is reduced from 4.91% (original Multistack) to 4.57%. Packing the stack pointers in two registers further reduces the performance overhead to 4.39%. The dedicated register approach is on average even 1% faster (3.91%). In case for each application the best performing countermeasure is chosen while requiring that no physical memory is wasted, the average performance overhead is reduced to 3.30%.

6 Discussion and ongoing work

Besides the possible implementations presented in section 4, another one was developed, but omitted due to page limits. As the technique described in sec-

Table 2. Running the telecom and consumer suite of the MiBench benchmark shows that the dedicated register approach, on average, outperforms the other approaches.

	unprot.	original		dedicated regs.		packed sp		fixed address	
application	sec	sec	%	sec	%	sec	%	sec	%
FFT	1.25 ± 0.01	1.43 ± 0.01	15.10%	1.33 ± 0.01	6.99%	1.36 ± 0.01	8.93%	1.36 ± 0.01	8.88%
IFFT	1.02 ± 0.01	1.21 ± 0.01	19.08%	1.11 ± 0.01	9.03%	1.13 ± 0.01	10.95%	1.13 ± 0.01	10.65%
gsm (enc.)	6.32 ± 0.02	6.33 ± 0.02	0.08%	6.32 ± 0.02	0.05%	6.32 ± 0.03	0.05%	6.33 ± 0.03	0.08%
gsm (dec.)	2.68 ± 0.02	2.68 ± 0.02	0.14%	2.68 ± 0.02	0.15%	2.68 ± 0.02	0.10%	2.68 ± 0.02	0.14%
jpeg (enc.)	0.34 ± 0.01	0.41 ± 0.01	19.31%	0.43 ± 0.01	25.53%	0.43 ± 0.01	26.19%	0.44 ± 0.01	27.12%
jpeg (dec.)	0.09 ± 0.01	0.09 ± 0.01	0.00%	0.09 ± 0.01	-0.34%	0.09 ± 0.01	0.67%	0.09 ± 0.01	0.22%
lame	26.42 ± 0.06	26.56 ± 0.08	0.53%	26.40 ± 0.07	-0.05%	26.64 ± 0.05	0.86%	26.53 ± 0.05	0.43%
mad	1.44 ± 0.01	1.48 ± 0.01	2.78%	1.50 ± 0.01	4.51%	—	—%	1.57 ± 0.01	8.95%
tiff2bw	0.94 ± 0.01	0.94 ± 0.01	-0.46%	0.94 ± 0.01	-0.11%	0.94 ± 0.01	-0.54%	0.94 ± 0.01	-0.27%
tiff2rgba	2.57 ± 0.02	2.56 ± 0.02	-0.13%	2.57 ± 0.02	-0.06%	2.57 ± 0.02	-0.03%	2.57 ± 0.02	0.03%
tiffdither	3.38 ± 0.02	3.46 ± 0.02	2.35%	3.41 ± 0.02	1.03%	3.42 ± 0.02	1.22%	3.46 ± 0.02	2.47%
tiffmedian	2.64 ± 0.02	2.64 ± 0.02	0.18%	2.64 ± 0.02	0.18%	2.63 ± 0.02	-0.14%	2.63 ± 0.02	-0.28%
average			4.91%		3.91%		4.39%		4.57%

tion 4.3, the stack pointers are stored in main memory but a reserved register stores their location. This eliminates the instruction to load the location in a register. However, its performance proved slightly worse in practice. We assume that the processor on our test was not able to schedule the instructions as well as the fixed address approach. More detailed information can be found in [19].

Using static analysis, many of the presented approaches can still be improved. At compile time it could be determined that some of the stacks will never be used by the application (see section 5.2). This knowledge could be applied to reserve less registers for the dedicated register approach. Combining this knowledge with a more guided estimation of the maximum size of each stack, the packed stack pointers technique could be modified to reduce the number of registers reserved and/or store less bits of each stack pointer.

Also a heuristic could be used in case registers are reserved to reduce the register pressure; functions may use these registers to store intermediate values, as long as their contents is restored before they are used by the function itself or one that it calls (in)directly.

The main goal was to eliminate the memory usage of the original Multistack implementation. Not only do all proposed implementations satisfy this goal, section 5.3 showed that on average they also reduce performance overhead. The dedicated registers approach not only outperforms all other implementations, but it can also be adapted easily for multithreaded applications, a situation not supported by the original approach.

7 Related work

Many countermeasures to protect applications against (stack-based) buffer overflow attacks have been developed during the last decades. In this section we briefly highlight the differences between our and existing approaches to the problem. A more elaborate overview can be found in [2, 15, 20].

One obvious way to defend applications against buffer overflow attacks is to add bound checks [3], however when implemented for C, their performance overhead is significantly larger than other countermeasures.

Another approach uses a type system and runtime checks to create safe languages where the existence of buffer overflow vulnerabilities are prevented. There are safe languages, referred to as safe dialects, that remain as close to C or C++ as possible. However, their performance overhead is also significant [4, 5].

The most often applied countermeasure in practice uses random data to detect buffer overflows or try to prevent their successful execution. StackGuard [6, 7] is an example of such a probabilistic countermeasure. It places a random value, called *canary*, in front of each return address on the stack. An attacker that overflows a stack-based buffer up to a return address, will modify the canary. The canary check at each function's epilogue will notice this modification and prevent the exploit before the return address is used.

However, by overwriting a pointer, an attacker is able to write to an arbitrary memory location, when the application dereferences the pointer for such an instruction. In case he/she is also able to control the written value, the execution path can be redirected to injected code, for example, by overwriting a return address [16]. Since the canary is not modified, StackGuard is not able to detect the attack. ProPolice [7] entangles the canary and the return address by taking the `xor` function from a random value and the return address. This allows the detection of any change to the return address.

Address space layout randomization (ASLR) [8, 9] randomizes the location of program data and code. As a result, redirecting execution to shellcode is hard.

While these last approaches are efficient, they rely on keeping memory locations secret. However, programs could also contain "buffer overreads" [10] or other vulnerabilities like format string vulnerabilities [21], which allow attackers to print out memory locations. Such memory leaking vulnerabilities could allow attackers to bypass this type of countermeasure.

Other countermeasures take advantage of hardware to protect against buffer overflows. Francillon et. al. [22] implement a similar approach as Multistack, but only two stacks are supported and the presence of specialized hardware is required. The techniques presented in this paper take advantage of a Memory Management Unit (MMU) to create guard pages, hardware that can also be used to facilitate additional countermeasures (e.g. non-executable memory [23, 8]) and already present in many modern processors.

8 Conclusion

The Multistack countermeasure effectively protects against stack-based buffer overflows, defeating many state-of-the-art attacks. Unfortunately, its straightforward port to the ARMv7 instruction set, does not reach the same efficiency as on the x86 platform and it suffers from high memory overhead.

In this paper we presented three approaches that reduce memory overhead up to 28%. In addition, each approach also reduces performance overhead. The best performing has an overhead of only 3.91% and supports multi-threaded applications, a situation not supported by the original Multistack implementation.

Acknowledgments. We thank Thomas Walter and Sven Lachmund for the discussions on this topic, and the feedback on an earlier draft of the paper. This research is partially funded by NTT Docomo Eurolabs, by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. Aleph1: Smashing the stack for fun and profit. Phrack **49** (1996)
2. Erlingsson, Ú., Younan, Y., Piessens, F.: Handbook of information and communication security. (2010)
3. Kendall, S.: Bcc: Runtime checking for C programs. In: Proceedings, The Association (1983) 5

4. Larus, J., Ball, T., Das, M., DeLine, R., Fähndrich, M., Pincus, J., Rajamani, S., Venkatapathy, R.: Righting software. IEEE software (2004) 92–100
5. Necula, G., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems (TOPLAS) **27**(3) (2005) 526
6. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (1998)
7. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks. Technical report, IBM Research Division, Tokyo Research Laboratory (June 2000)
8. Team, P.: Documentation for the PaX project. Homepage of The PaX Team (2003)
9. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. Proceedings of the 12th USENIX Security Symposium (August 2003) 105–120
10. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: EUROSEC '09: Proceedings of the Second European Workshop on System Security, New York, NY, USA, ACM (March 2009) 1–8
11. National Institute of Standards and Technology: National vulnerability database statistics. (2009) http://nvd.nist.gov/statistics.cfm.
12. Younan, Y., Philippaerts, P., Piessens, F., Joosen, W., Lachmund, S., Walter, T.: Filter-resistant code injection on ARM. In: Proceedings of the 16th ACM conference on Computer and communications security, ACM (2009) 11–20
13. : National vulnerability database (cve-2006-4131) (2006)
14. Younan, Y., Pozza, D., Piessens, F., Joosen, W.: Extended protection against stack smashing attacks without performance loss. ACSAC (2006)
15. Younan, Y., Joosen, W., Piessens, F.: Code injection in c and c++ : A survey of vulnerabilities and countermeasures. Technical report, Departement Computerwetenschappen, Katholieke Universiteit Leuven (2004)
16. Bulba, Kil3r: Bypassing stackguard and stackshield. Phrack Magazine **0xa**(0x38) (January 2000)
17. Ergin, O., Balkan, D., Ghose, K., Ponomarev, D.: Register packing: Exploiting narrow-width operands for reducing register file pressure. In: Proceedings of the 37th International Symposium on Microarchitecture. (2004)
18. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE 4th annual Workshop on Workload Characterization. Volume 131. (2001) 184–193
19. Strackx, R.: Protecting mobile devices against stack-based buffer overflows. Master's thesis, Katholiek Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven (June 2009)
20. Younan, Y.: Efficient countermeasures for software vulnerabilities due to memory management errors. PhD thesis, Katholiek Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven (May 2008)
21. Shankar, U., Talwar, K., Foster, J., Wagner, D.: Detecting format string vulnerabilities with type qaualifiers. In: Proceedings of the 10th conference on USENIX Security Symposium-Volume 10, USENIX Association (2001) 16
22. Francillon, A., Perito, D., Castelluccia, C.: Defending Embedded Systems Against Control Flow Attacks
23. : Non-executable stack patch (1998) http://www.openwall.com.